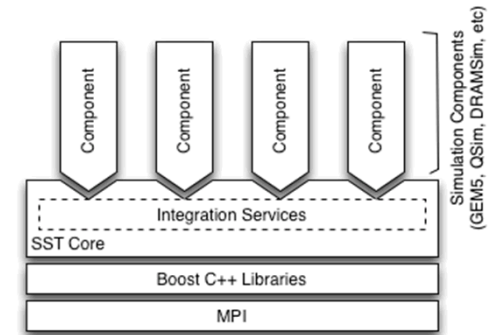
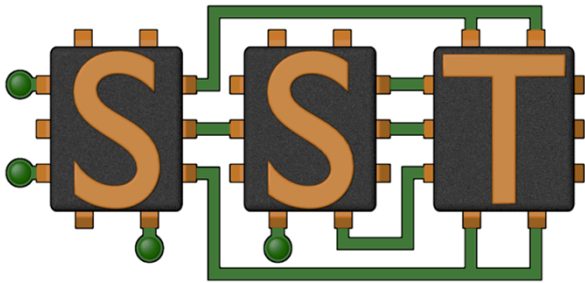


Exceptional service in the national interest



Structural Simulation Toolkit

Lunch & Learn

September, 2015

Branden Moore (5638)

SST Lunch & Learn

- Introduction to SST
- Examples of use at Sandia
- Framework Overview
- Building a Simulator

Why SST?

- Problem: Simulation is slow
 - Tradeoff between accuracy and time to simulate
 - Many simulators are serial, unable to simulate very large systems
- Problem: Lack of simulator flexibility
 - Tightly-coupled simulations: faster but difficult to modify
 - Difficult to simulate at different levels of accuracy

***The Structural Simulation Toolkit:
A parallel, discrete-event simulation framework
focused on scalability and flexibility.***

SST Key Features

- **Parallel**
 - Built from the ground up to be scalable
 - Demonstrated scaling to 512+ processors, Millions of Components
 - Supports both MPI and Thread-based parallelism

- **Flexible**
 - Enables “mix and match” of simulation components
 - Timescale agnostic (femtoseconds to years)
 - Customize tradeoff between accuracy and simulation time
 - E.g., cycle-accurate network with trace-driven endpoints
 - Non-viral, Open Source license

- **Mature, but active**
 - Version 5.1 Released in September, 2015
 - Current SVN/git history back to 2009
 - Over 117k SLOC (Core alone: ~20k SLOC)

SST Project Collaboration

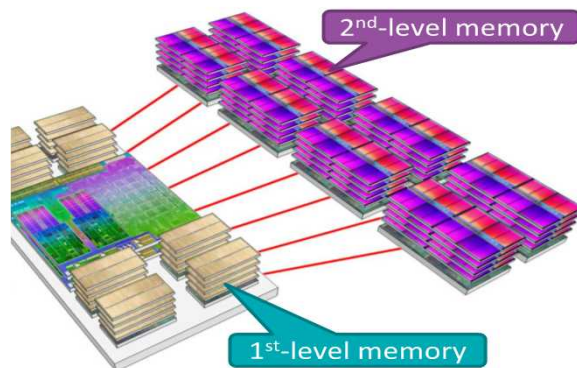


Primary development: Sandia 1420

EXAMPLE USE CASES

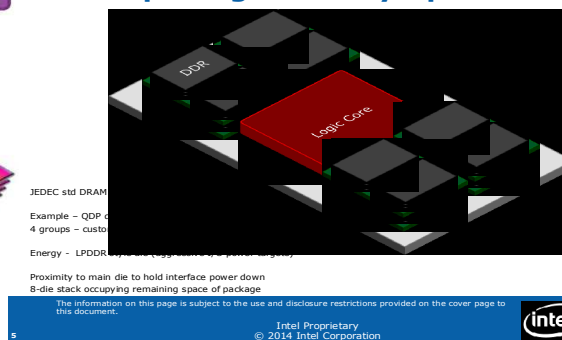
Case #1: Multi-level memory

- Future memory systems will be **Multi-Level Memory**
- MLM can potentially offer more “usable” bandwidth, less cost
- Challenges:
 - **substantial** software and hardware (co-)design
 - **no** “one size fits all”
- SST can explore HW & SW organization

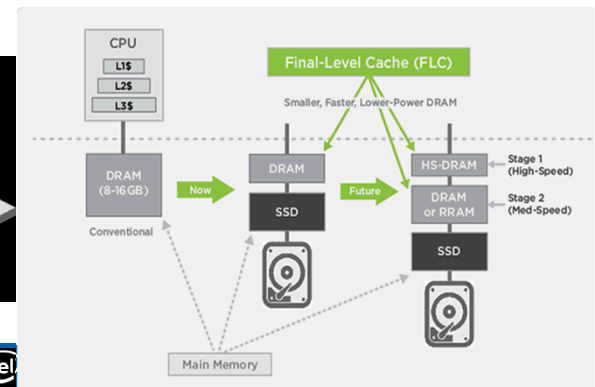


AMD

On package Memory Option



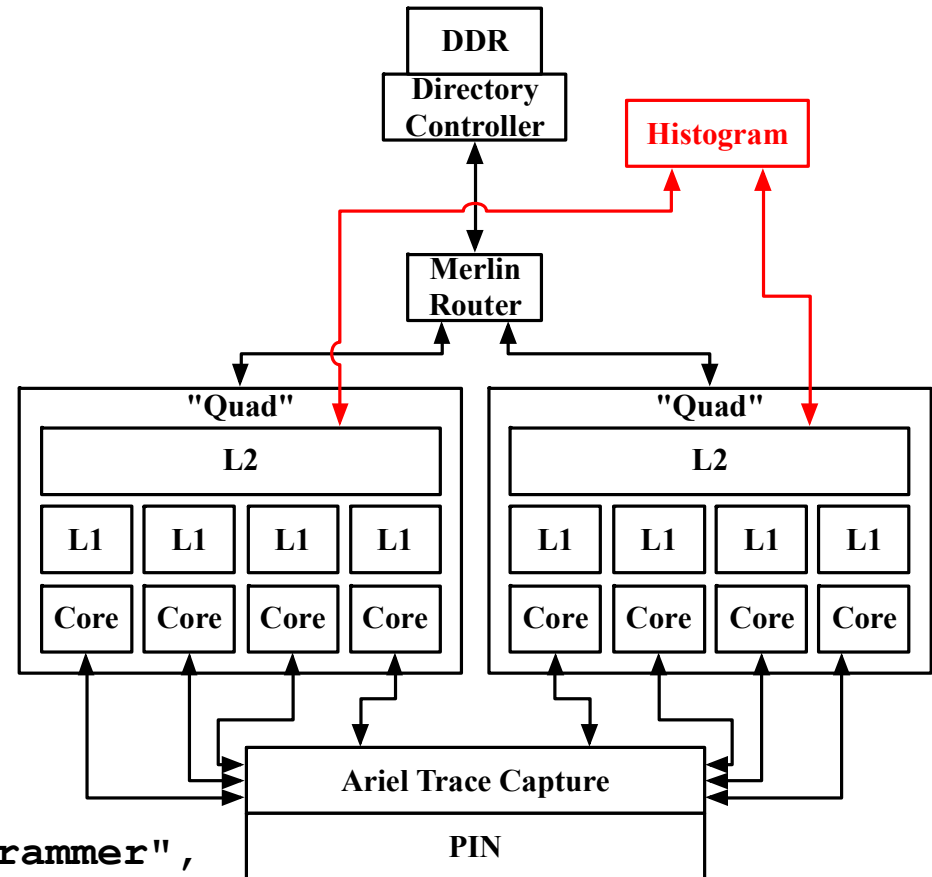
Intel



Marvell

Analyzing Memory Accesses

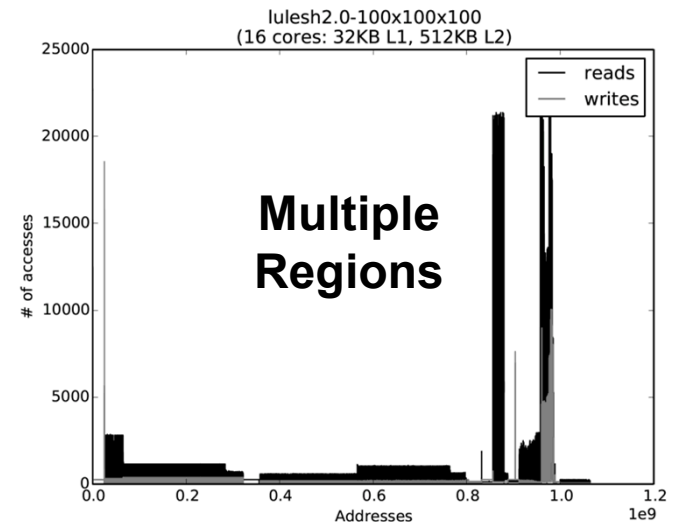
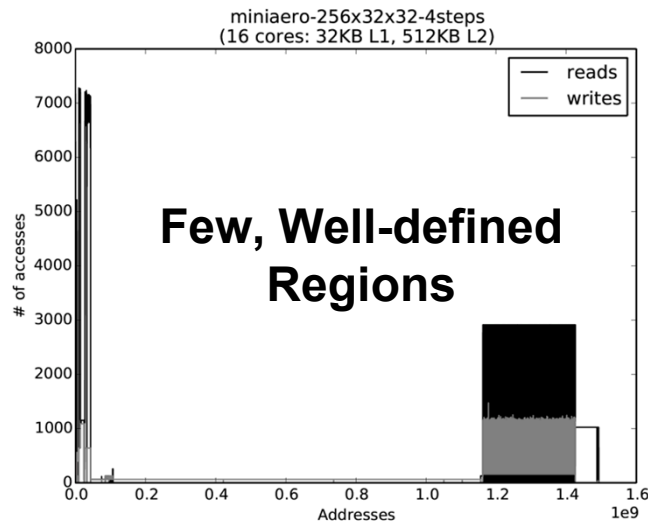
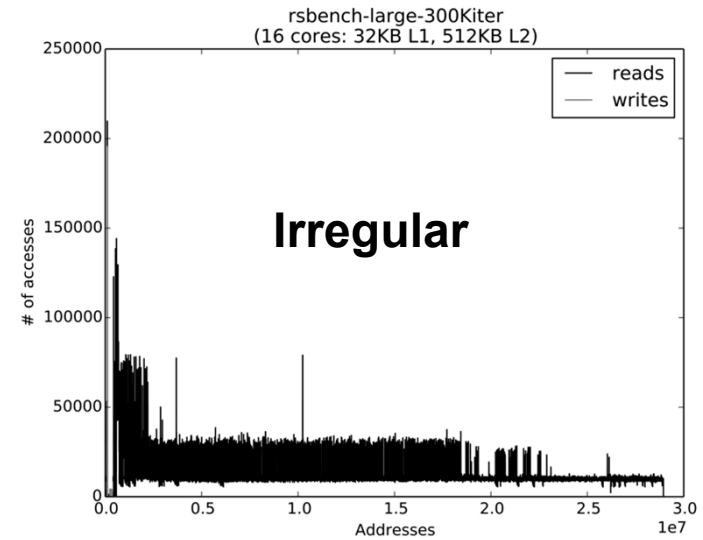
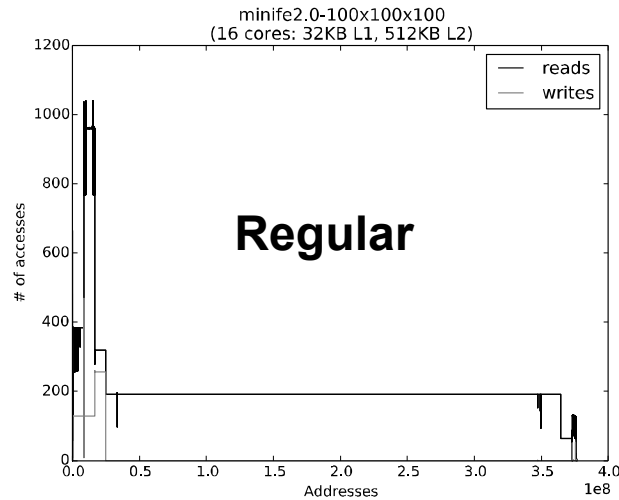
- Capture post-cache accesses
- Setup:
 - “Quads” of 4 cores
 - Histogram generator implemented as a prefetcher



```
12SnoopParams = {  
  "prefetcher": "cassini.AddrHistogrammer",  
  "prefetcher.histo_bin_width": 4096,  
  "prefetcher.heap_begin": "1 GiB",  
  "prefetcher.heap_end": "9 GiB"  
}
```


Analysis: Diverse Patterns

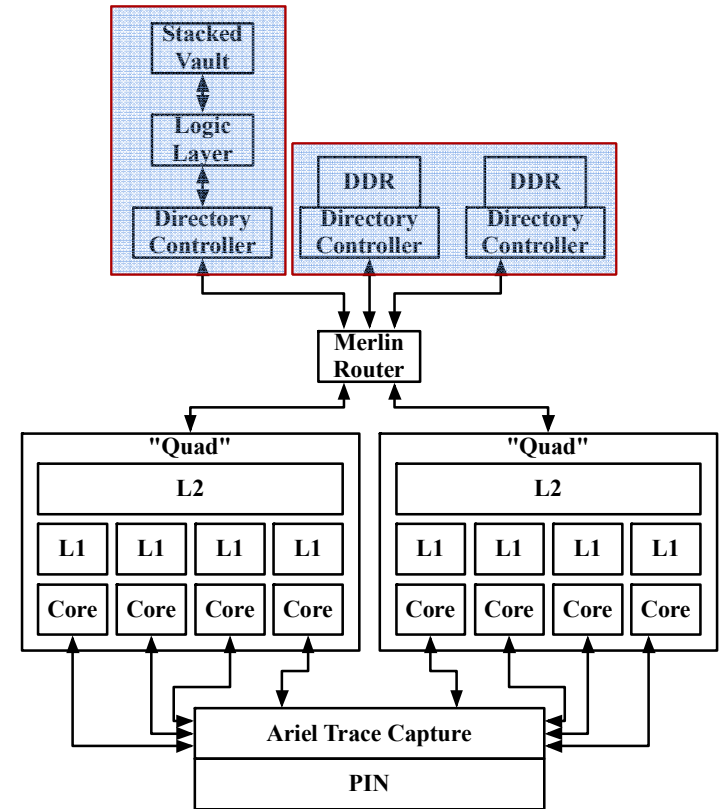
Physical address histograms



Multi-Level Memory Simulation Sandia National Laboratories

- Multiple memory types:
 - DDR DRAM (DramSim)
 - HMC-like Stacked Memory (VaultSim)
 - NVRAM (NVDIMMSim)
- Addresses can be interleaved, or blocked between memory types

```
dc.addParams({  
  "addr_range_start": start_pos,  
  "addr_range_end": end_pos,  
  "interleave_size": interleave_size/1024,  
  "interleave_step": interleave_step,  
  "entry_cache_size": 128*1024,  
  "clock": memclock,  
  "network_address": netPort  
})
```



MLM Explorations

- Analysis of application memory use distribution
- Quick exploration of “Naïve” address assignment, capacity ratios on performance
- Not shown: Feedback results from histograms to determine address assignment

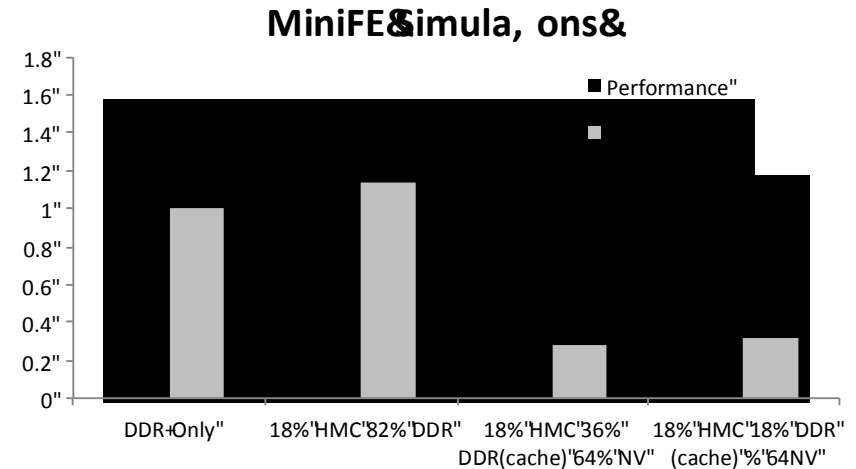


Figure 5: MiniFE Simulation results

Case #2: Network

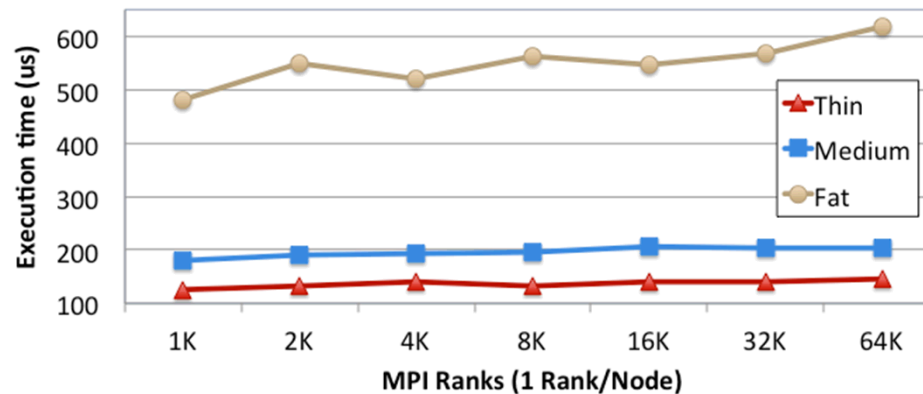
- What is the network latency achieved by different platforms during a 3D halo exchange?
 - *Halo exchange*: Exchange boundary data with neighbors
 - Platform 1: “Fat” nodes – Eight 20TF/s cores per node
 - Platform 2: “Medium” nodes – Two 20TF/s cores per node
 - Platform 3: “Thin” nodes – One 10TF/s core per node
- Evaluate for 1K to 64K participating nodes
- Evaluate at three different link bandwidths
 - 12.5GB/s, 50GB/s, 125GB/s

Network: Simulation setup

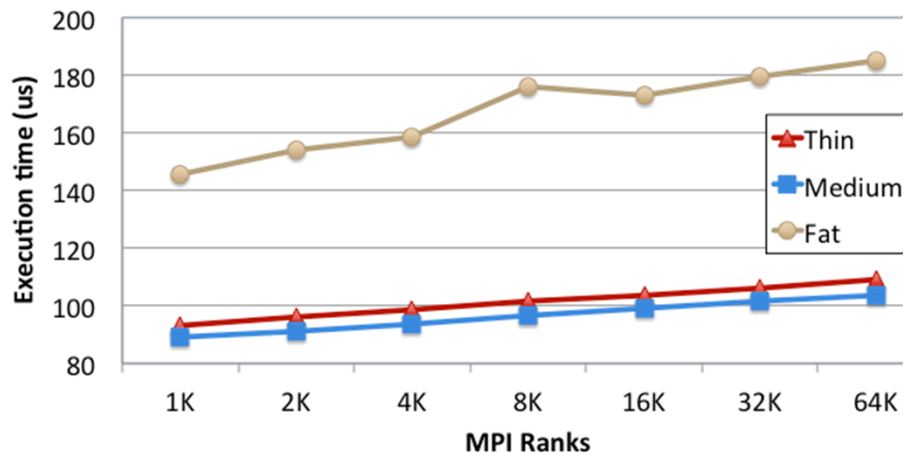
- Use *Ember* to model nodes
 - Lightweight model focused on communication pattern
 - Estimates compute time using the node's FLOPS
 - Detailed model of communication
 - Enables scaling the simulated system to a larger number of nodes
 - Compared to a detailed processor model + memory model
- Use *Firefly* to model the NIC
- Use *Merlin* to model the network
 - Detailed, cycle-accurate models for network (routers, links, etc.)

Scaling with Bandwidth

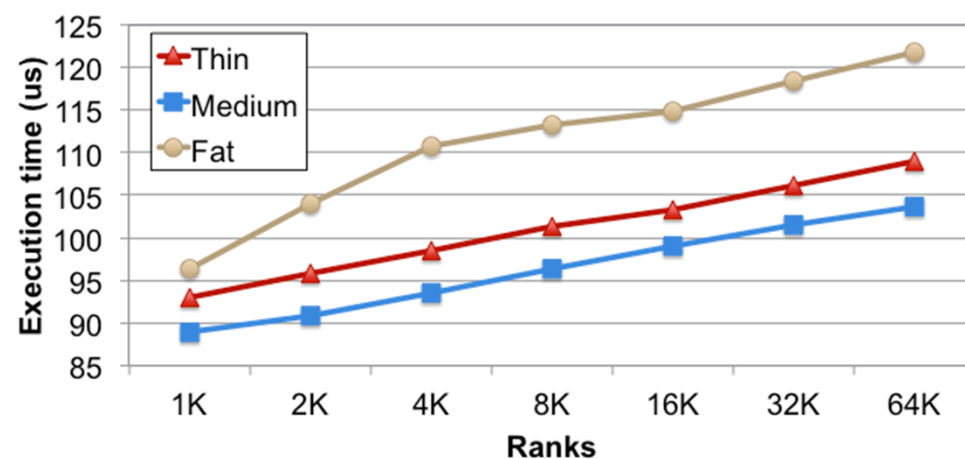
Link Bandwidth = 12.5GB/s



Link Bandwidth = 50GB/s



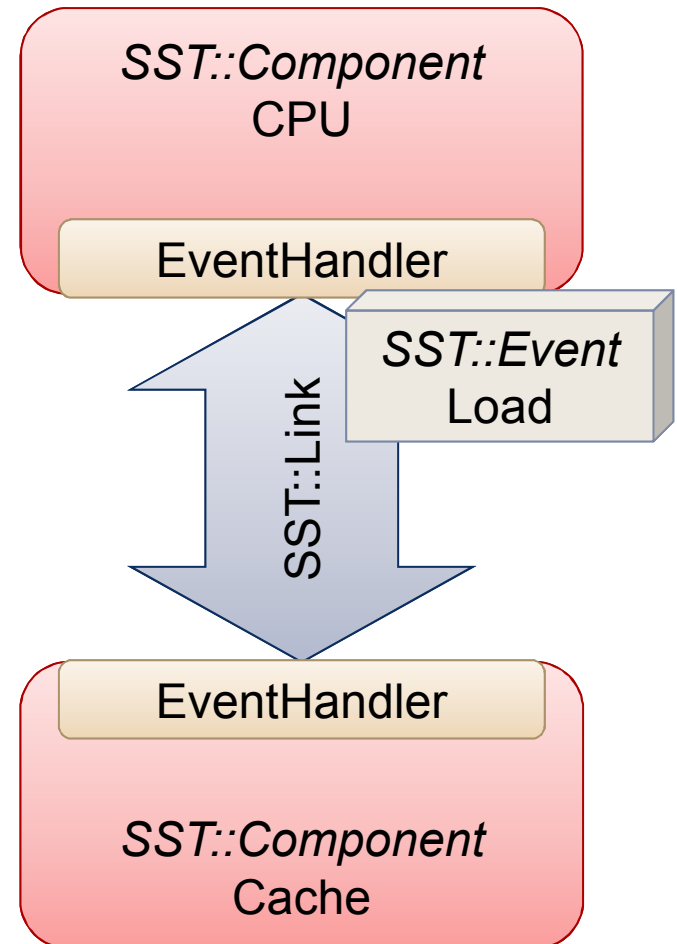
Link Bandwidth = 125GB/s



SST CORE: FRAMEWORK FOR PARALLEL SIMULATION

Key Concepts

- `SST::Component`
 - Simulation model
- `SST::Link`
 - Communication path between two components
- `SST::Event`
 - A discrete event
- `SST::SubComponent`
 - Add functionality to Components
- `SST::Module`
 - Add functionality to framework
- Element Libraries
 - Contain Components and Modules in a Shared Library format
 - Included manifest documents components, features, parameters and interactions



Component

- Basic building block of a simulation model
 - E.g., processor, cache, network router, etc.
- Performs the actual simulation
 - Zero Simulation time while operating
 - Time advances on events
 - Can send events to itself for timing purposes
- Events are issued on components
 - Can register Clock Events
 - Receive events over links
- Communicate with other components
 - Components define ports, links connect ports between components
 - Polled: Register a clock handler to poll the link
 - Interrupt: Register an event handler to be called when an event arrives
 - Both: Receive events on interrupt, send events on clock
- SubComponents and Modules provide additional functionality

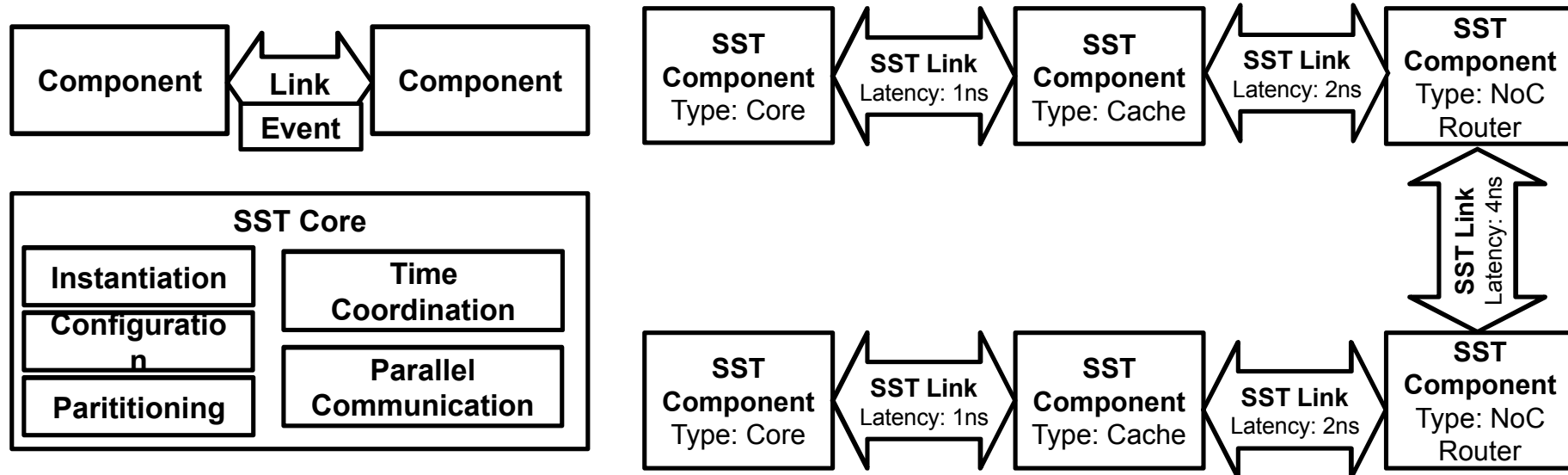
Link

- Connects two components
 - Connect a specific “Port” on component A to a “Port” on component B
- The ONLY mechanism by which components communicate
 - Necessary for parallel simulation
- Has a minimum, non-zero latency for communication
 - Except self-links
 - Except during initialization
- Transparently handles any MPI communication



- Unit of communication between two components
 - Packet format is up to the communicating components
 - Base class is SST::Event
 - Has a delivery time
 - Typically calculated from a link
 - Can specify priorities.
- Some standardized interfaces
 - Facilitate “mix and match” capability
 - sst/core/interfaces/
 - Memory (simpleMem)
 - Defines commands & event format for communication with memory
 - Network (simpleNetwork)
 - Defines a header for events sent through a network component

SST's discrete-event algorithm



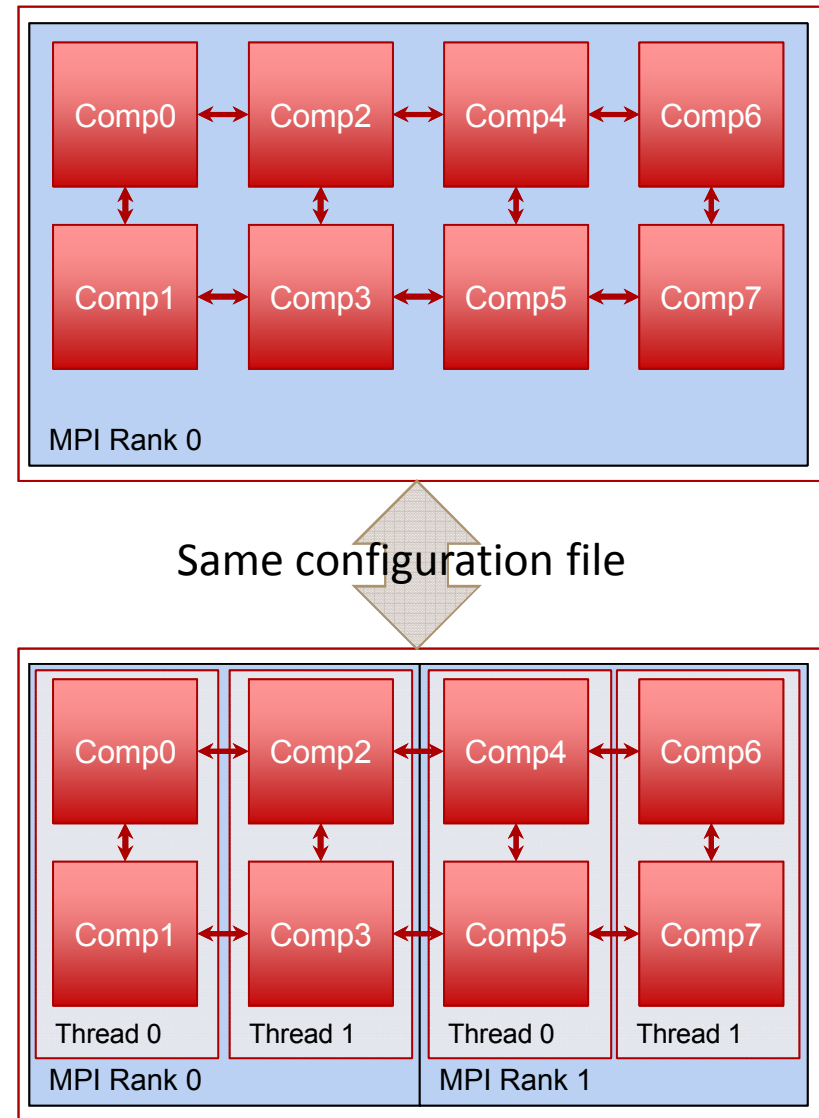
- Simulations are comprised of **components** connected by **links**
- **Components** interact by sending events over **links**
- Each **link** has a *minimum* latency
- **Components** can load **subComponents** and **modules** for additional functionality

Simulation lifecycle

- Birth
 - **Create graph** of components using Python configuration file
 - **Partition** graph and assign components to parallel ranks
 - **Instantiate** components
 - **Connect** components via links
 - **Initialize** components using their `init()` functions
 - **Setup** components using their `setup()` functions
- Life
 - **Send events**
 - **Synchronization** between parallel ranks (hidden from user)
 - **Manage clock and event handlers**
- Death
 - **Finalize** components using their `finish()` functions
 - **Output** statistics
 - **Cleanup** simulation, delete components

SST in parallel

- SST was designed from the ground up to enable scalable, parallel simulations
 - Conservative, Distance-based Optimization strategy
- Components are distributed among MPI ranks and Threads
- Links enable parallelism
 - Components only communicate via links
 - Specified link-latency determines synchronization rate
 - Transparently handle any MPI communication & thread synchronization
- Multiple partitioning strategies
 - Linear, RR, “Simple”, Zoltan-based
 - Simulation writer can provide own partitioning



SST FOR BUILDING SIMULATORS

What makes a Simulator?

- Simulation Engine

- Handle events, communication between components

SST Core

- Simulation Models

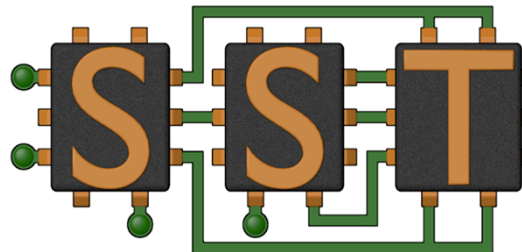
- Perform actual simulation / emulation

Element Libraries

- System Construction

- Connect models together, configure the system to simulate

Python Module



SST as a Framework

- Framework Provides:
 - Mature PDES Engine
 - Serial or parallel (Multi-Thread and Multi-Rank)
 - Optimizations for very large simulators
 - “Sleeping” components
 - Shared Memory Lookup Tables
 - Memory Management
 - API for building simulation components
 - Manifests for documentation
 - Utilities for common simulation challenges
 - Parameters/Configuration, Statistics gathering, repeatable PRNG
 - Python Module for building Simulators
 - Configure Components and Links via Python scripts
 - Supports embedding additional modules in Element Libraries

Sandia-provided Models

- Processors
 - Ariel – PIN-based
 - Prospero – Trace-based
 - Miranda – Pattern-based
- Memory
 - MemHierarchy – Caches, “Main Memory”
 - VaultSimC - Stacked memory
 - Cassini – Cache prefetchers
- Network driver
 - Ember – Pattern-based
 - Firefly – communication protocols
 - Hermes - MPI-like driver interface
 - Zodiac – trace-based
- Network models
 - Merlin – HPC Network simulator
- Other
 - Scheduler
 - simpleElementExample

Building custom Components

- Create a C++ class which inherits from 'SST::Component'
 - Constructor:
 - Will be passed an SST::Params object with configuration information
 - Registers any clocks and their event handlers
 - Configures links with event handlers
 - Initialization phase:
 - Can pass data to peers (pre-fill memory, establish global state)
 - Run time:
 - Respond to events (clocks, links)
 - Simulate target model
 - Send new events
 - Record statistics
- Build an Element Library to describe this component (or group of components)

Element libraries

- Collection of related components, subComponents, and modules
- Includes a manifest
 - Declares how to create new component instances
 - Documents Components, Parameters, Ports and Statistics
- Implemented as a Shared Library with a well-known symbol for the Manifest
- SST comes with many built-in libraries from Sandia
 - Processors, memory, network, etc.
 - Tested for inter-library compatibility

Building a Simulator

- SST provides a Python Module with an API to connect and configure simulation components to build a simulator
 - `import sst`
- At run time:
 - `sst 'mySim.py'`
 - SST will launch Python interpreter, and execute `'mySim.py'`
 - Script should instantiate all needed components, and provide their parameters
 - Command-line arguments can be passed to the interpreter
 - `--model-options "-a -b 5"`
 - Normal python interpreter functionality available
 - `import getopt`
 - `import antigravity`

```
import sst

# Define the simulation components
comp_msgGen0 = sst.Component("msgGen0",
    "simpleElementExample.simpleMessageGeneratorComponent")
comp_msgGen0.addParams({
    "outputinfo" : 0,
    "sendcount"  : 10000,
    "clock"      : "1MHz"
})
comp_msgGen1 = sst.Component("msgGen1",
    "simpleElementExample.simpleMessageGeneratorComponent")
comp_msgGen1.addParams({
    "outputinfo" : 0,
    "sendcount"  : 100000,
    "clock"      : "1MHz"
})

myLink = sst.Link("myLink")
myLink.connect( (comp_msgGen0, "remoteComponent", "1us"),
    (comp_msgGen1, "remoteComponent", "1us"))
```

Python Module – Defining Components

- **Define:** `sst.Component("name", "type")`
- **Configure:** `addParams ({ "parameter" : value, ... })`

Instance name

Element Library

Component type

```
network = sst.Component("router", "merlin.hr_router")
network.addParams({
    "xbar_bw" : "51.2GB/s",
    "link_bw" : "25.6GB/s",
    "num_ports" : 4,
    "flit_size" : "72B",
    "topology" : "merlin.singlerouter",
    "id" : 0,
    "input_buf_size" : "2KB",
    "output_buf_size" : "2KB"
})
```

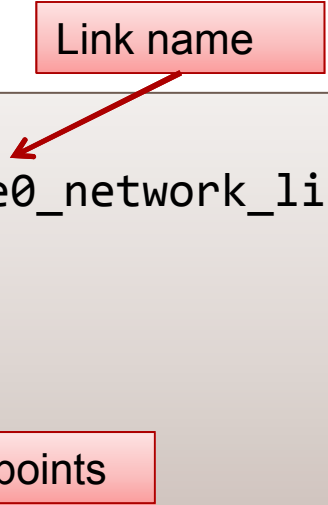
Parameters

Python Module – Defining Links

- Example: Connect an L2 cache (l2cache0) to network
 - **Create a link:** `sst.Link("name")`
 - **Define link endpoints:** `connect(endpoint1, endpoint2)`
 - Endpoint is defined as: (Component, Port, Latency)
 - Note: Latencies of the two endpoints can differ

```
...
l2cache0_network_link = sst.Link("l2cache0_network_link")
...
l2cache0_network_link.connect(
    (l2cache0, "directory", "500ps"),
    (network, "port0", "1ns")
...

```



SSTInfo: Getting component info

- Prints parameters, port names, statistics, ...

Optionally filter for a specific component

```
$ sstinfo memHierarchy.Cache  
PROCESSED 25 .so (SST ELEMENT) FILES FOUND IN DIRECTORY /home/sst/build/lib/sst  
Filtering output on Element.Component = "memHierarchy.Cache"
```

```
=====
```

ELEMENT 18 = memHierarchy (Cache Hierarchy)
COMPONENT 0 = Cache [MEMORY COMPONENT] (Cache Component)
NUM PARAMETERS = 32
 PARAMETER 0 = cache_frequency (Clock frequency with units. For L1s, this is usually the same as the CPU's frequency.) [REQUIRED]

"REQUIRED" or default value

...
PARAMETER 21 = network_bw (Network link bandwidth.) [1GB/s]

Parameter

Definition

NUM PORTS = 4

Port name

...
PORT 3 [1 Valid Events] = directory (Network link port to directory)
 VALID EVENT 0 = MemHierarchy.MemRtrEvent

Definition

...
NUM STATISTICS = 32

Type of event(s) used on the link

Running SST

- Usage: `sst [options] configFile.py`
- Common options:

<code>-v --verbose</code>	Print verbose information during runtime
<code>--debug-file <filename></code>	Send debugging output to specified file (default: <code>sst_output</code>)
<code>--add-lib-path <dirname></code>	Add <code><dirname></code> to search path for element libraries
<code>--heartbeat-period <period></code>	Every <code><period></code> time, print a heartbeat message
<code>--partitioner <zoltan self simple rrobin linear lib.partition.name></code>	Specify the partitioning mechanism for parallel runs
<code>--model-options "<args>"</code>	Command line arguments to send to the Python configuration file
<code>--output-partition <filename></code>	Write partitioning information to <code><filename></code>
<code>--output-dot <filename></code>	Output a graph representing the configuration in "Dot" format to <code><filename></code>

Finally...

- SST Wiki: <http://www.sst-simulator.org/>
 - Downloading, installing, and running SST
 - Element libraries and external components
 - Guides for extending SST
 - Information on APIs
 - Information about current development efforts
- Mailing lists:
 - ***sst-user***: For questions on building, compiling, extending, and using SST
 - ***sst-developer***: For questions on developing SST components
 - ***sst-announce***: Release announcements
 - ***sst-commit***: Notification of commits to the SVN repository
 - Subscribe via the wiki

THANK YOU

Backup

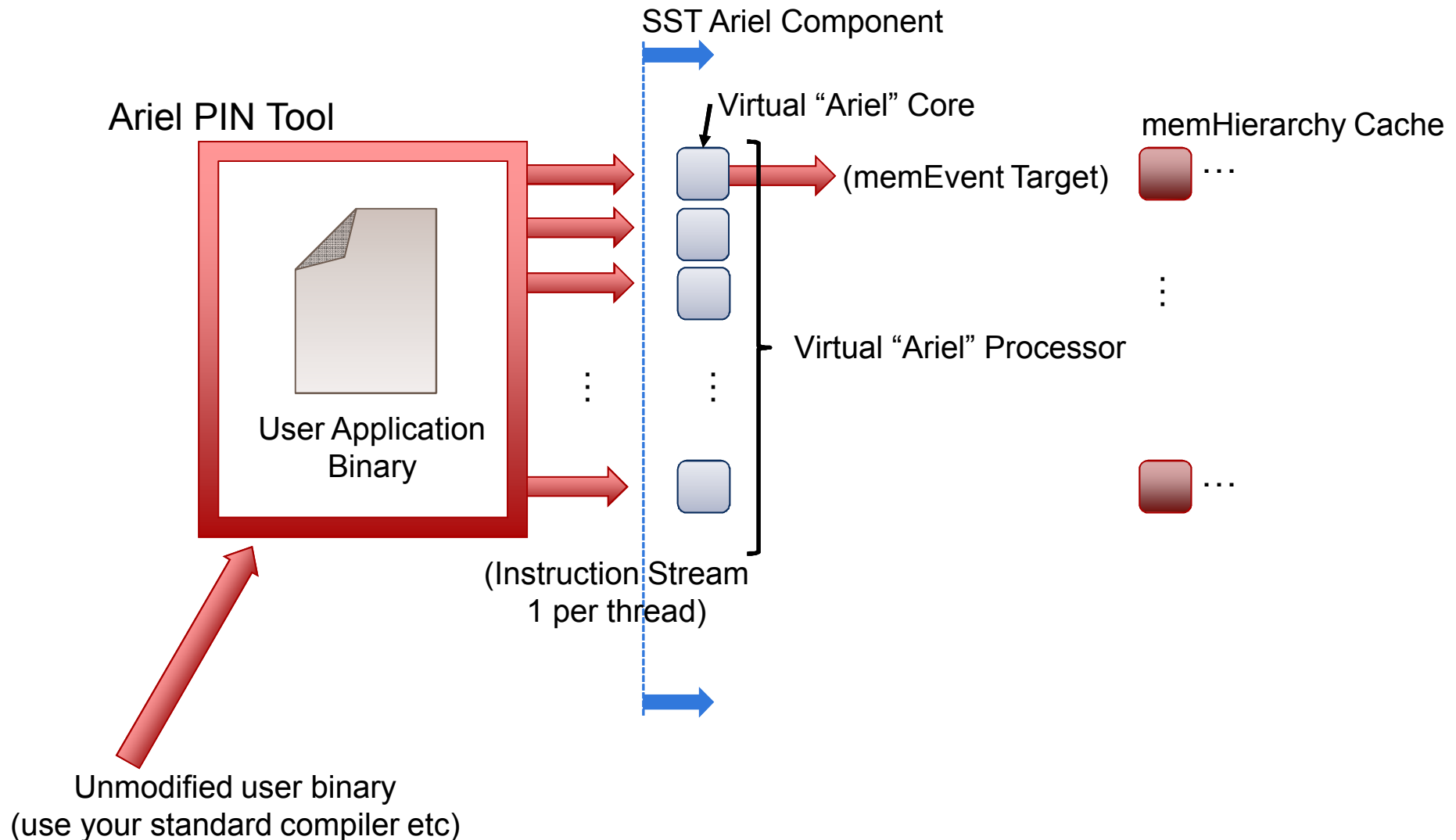
Ariel: PIN-based processor

- Lightweight processor core model
- Uses Intel's PIN tools and XED decoders to analyze binaries
 - Runs x86, x86-64, SSE/AVX, etc. compiled binaries
 - Supports fixed thread count parallelism (OpenMP, Qthreads, etc.)
- Passes information to virtual core in SST
- Implements SST's memory interface to interact with a memory model

Ariel: The tradeoff

- Pros:
 - Faster than cycle-accurate processor models (e.g., *Gem5*)
 - Reasonable approximation for studies on memory system performance
 - Especially for heavily memory-bound applications
 - Reasonable model of thread interactions
- Cons
 - Slower than trace/pattern-based processor models
 - Does not give cycle-reproducible results
 - Use of threads can disturb reproducibility
 - Non-deterministic results
 - Not compatible with non-x86 binaries

Ariel: Architecture



Ariel: Details

- Ariel's virtual cores
 - Instruction information currently limited to memory ops or instructions with no memory operands
 - Clocked: Reads instruction stream in chunks but processes on clock
 - Back pressure from FIFO halts real binary execution
 - Does not maintain dependence order or register locations (yet)!
 - Performs a TLB mapping of virtual-to-physical addresses
- Key user knobs
 - Memory ops issued/cycle
 - Load/store queue size
- Memory interface
 - Generates memEvents which can be sent to a cache model
 - Tracks basic statistics (request counts, split-cache line loads, etc.)

Prospero: Trace-based processor Sandia National Laboratories

- Trace-based processor model
 - Reads memory ops from a file and passes to the simulated memory system
 - “Single core” but can use multiple trace files to emulate threaded or MPI-style applications
 - Supports arbitrary length reads to account for variable vector widths
 - Performs “first touch” virtual to physical mapping
- Comes with Prospero Trace Tool to generate traces
 - Or can generate your own and translate to Prospero’s format

Prospero: The tradeoff

■ Pros

- Faster than Ariel and Gem5
 - Provided you can get a trace
- Good for heavily memory-bound applications
- Reasonable approximation to memory system performance

■ Cons

- Traces can be very large
 - Requires good I/O system to store and read the trace
- Traces are less flexible than actual execution
 - Capture a single execution stream using a single application input

Miranda: Pattern-based processor

- Extremely light-weight processor model
 - Generates specific memory address patterns
- Current patterns
 - Strided accesses (single stream)
 - Forward and reverse strides
 - Random accesses
 - GUPS
 - STREAM benchmark
 - In-order & out-of-order CPU
 - 3D stencil
 - Sparse matrix vector multiply (SpMV)
 - Copy (~array copy)

Miranda: The tradeoffs

- Pros
 - Very lightweight – no binary, no trace
 - Good for applications whose address patterns are predictable
 - E.g., not much pointer-chasing
- Cons
 - Need a generator for the memory pattern of interest
 - Requires a good understanding of the pattern

MemHierarchy: Memory system



- Cycle-accurate cache and memory simulation
 - Inter- and intra-socket coherence
 - Multiple main memory models
- Highly configurable
 - Can model any number of caches (L10s!)
 - Arbitrary topologies, multiple memories
 - Single- and multi-socket configurations
- Capable of modeling modern memory hierarchies
 - Intel core i7, Xeon Phi
 - Arm Cortex A8, A7, A15, A53, A57
 - SPARC T6

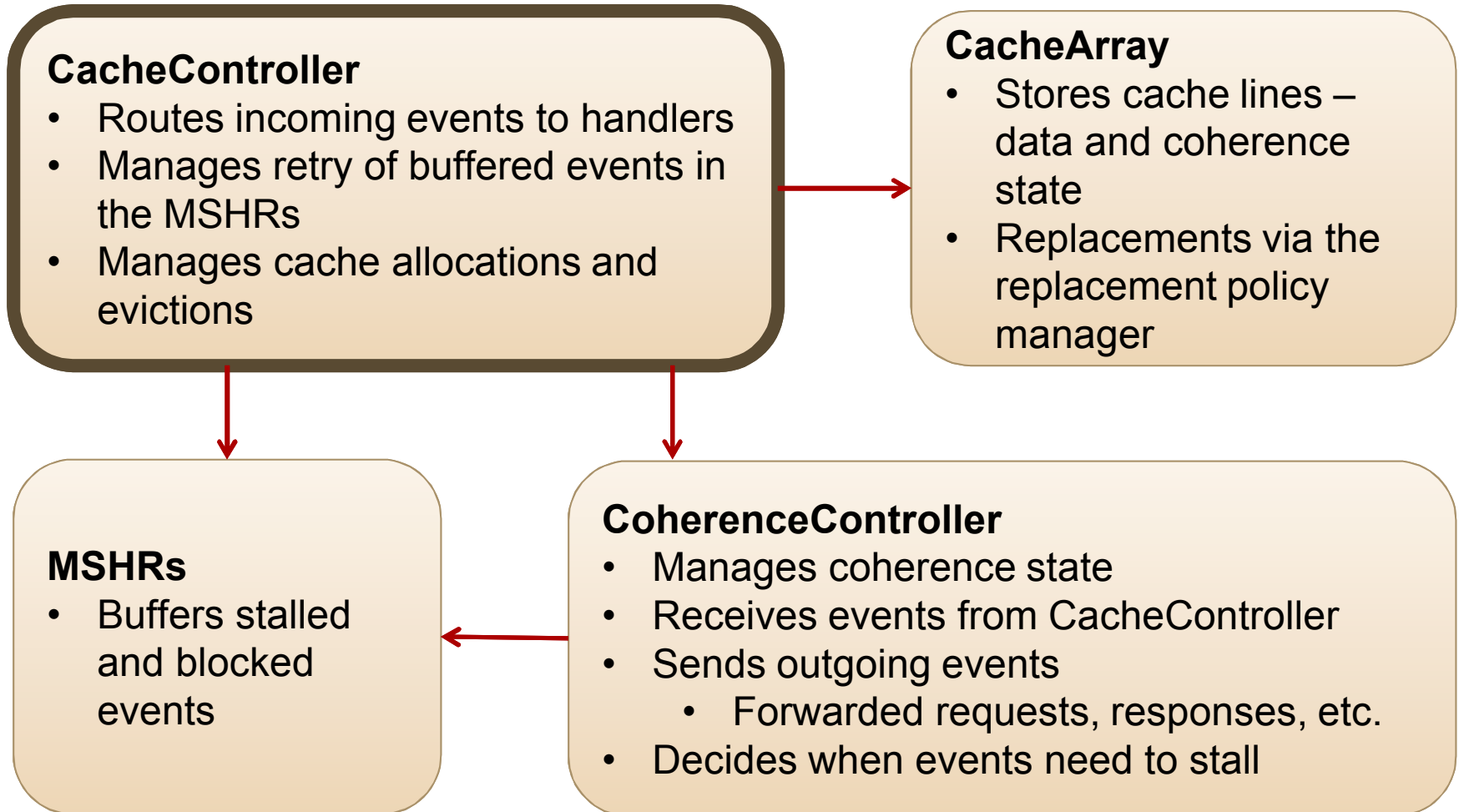
MemHierarchy: Components

- Cache
 - Includes coherence protocols (MSI, MESI, etc.)
- Bus
- Directory controller
 - Inter-socket coherence
- Memory controller
 - Backs up simulated memory, interfaces with memory backends
- Memory backends
 - Main memory simulators for DRAM, stacked DRAM, NVRAM, etc.
- TrivialCPU & StreamCPU
 - Very simple memory request generators for testing

MemHierarchy: Caches

- Store actual data
- Set associative, configurable replacement policies
 - LRU, LFU, Random, MRU, NMRU (not MRU)
- Use MSHRs to buffer outstanding requests
- Can communicate via a direct link or over a bus or network
 - Implements simpleNetwork interface via the “MemNIC” module
- Can model a single shared cache or multiple cache slices
- Handles atomics, LLSC, non-cacheable requests, etc.
- Prefetch capability by using the *Cassini* element library

MemHierarchy: Cache structure



MemHierarchy: Main memory



- MemoryController
 - Contains a 'backing store' for simulated data
 - Can communicate over a network or via a direct link with a cache or directory
 - Interfaces with multiple memory backends
- Available backends
 - SimpleMem – basic read/write with associated latencies
 - DRAMSim2 – DRAM (external)
 - NVDIMMSim – Non-volatile memory (e.g., Flash) (external)
 - HybridSim – non-volatile memory with a DRAM cache (external)
 - VaultSimC – stacked DRAM

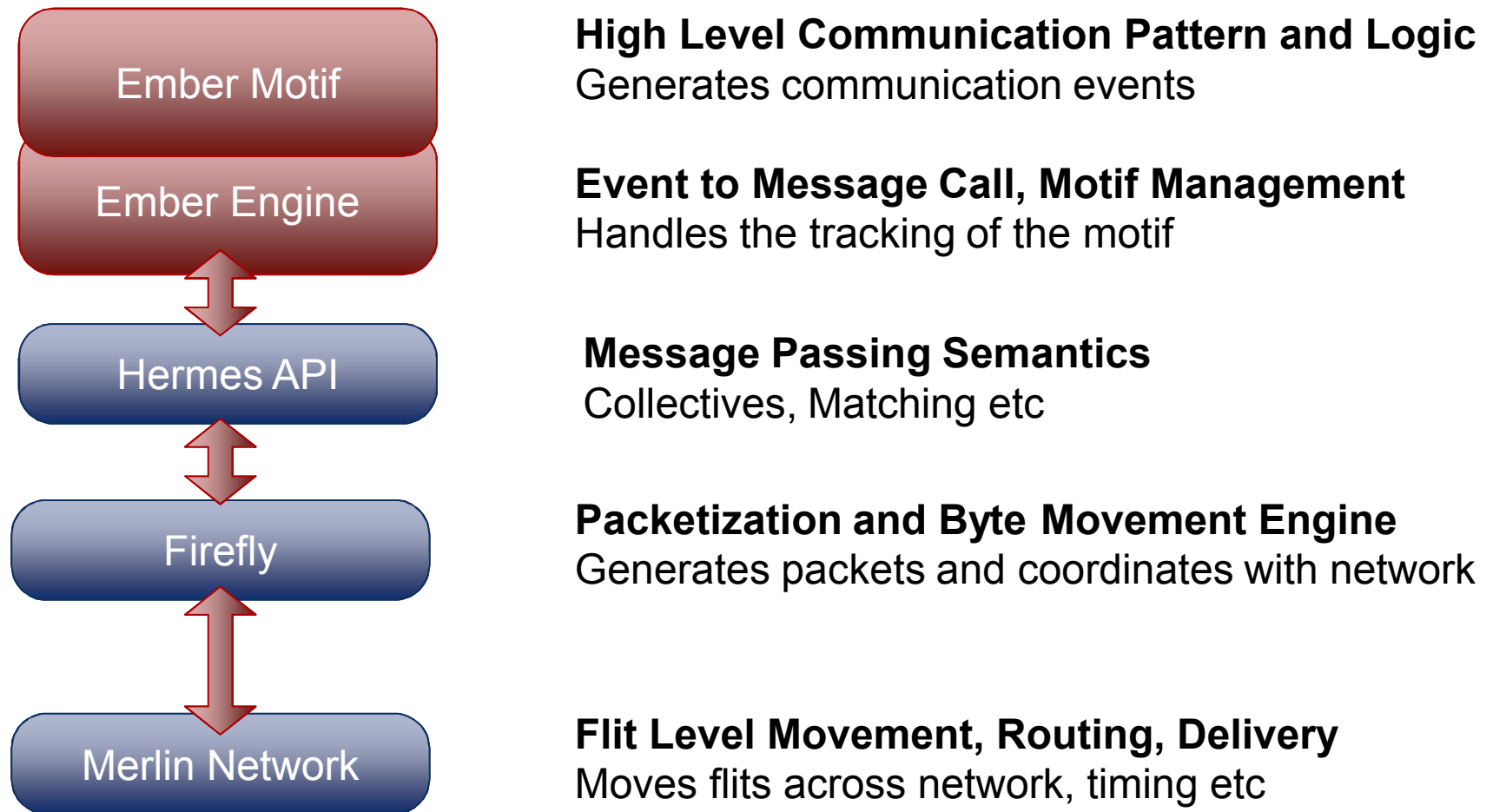
Merlin: Network simulator

- Low-level, flexible networking components that can be used to simulate high-speed networks (machine level) or on-chip networks
- Capabilities
 - High radix router model (*hr_router*)
 - Topologies – mesh, n-dim tori, fat-tree, dragonfly
- Many ways to drive a network
 - Simple traffic generation models
 - Nearest neighbor, uniform, uniform w/ hotspot, normal, binomial
 - *MemHierarchy*
 - Lightweight network endpoint models (*Ember* – coming up next)
 - Or, make your own

Ember: Network traffic generator Sandia National Laboratories

- Light-weight endpoint for modeling network traffic
 - Enables large-scale simulation of networks where detailed modeling of endpoints would be expensive
- Packages patterns as *motifs*
 - Can encode a high level of complexity in the patterns
 - Generic method for users to extend SST with additional communication patterns
- Intended to be a driver for the Hermes, Firefly, and Merlin communication modeling stack
 - Uses Hermes message API to create communications
 - Abstracted from low-level, allowing modular reuse of additional hardware models

Ember: Overview



Ember: Motifs

- Motifs are lightweight patterns of communication
 - Tend to have very small state
 - Extracted from parent applications
 - Models as an MPI program (serial flow of control)
 - Many motifs acting in the simulation create the parallel behavior
- Example motifs
 - Halo exchanges (1, 2, and 3D)
 - MPI collections – reductions, all-reduce, gather, barrier
 - Communication sweeping (Sweep3D, LU, etc.)

Ember: Motifs (continued)

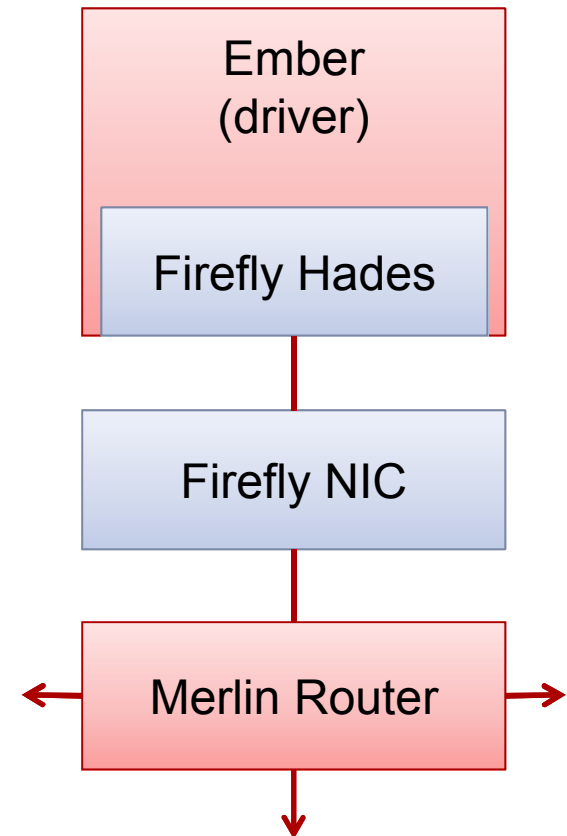
- The EmberEngine creates and manages the motif
 - Creates an event queue which the motif adds events to when probed
 - The Engine executes the queued events in order, converting them to message semantic calls as needed
 - When the queue is empty, the motif is probed again for events
- Events correspond to a specific action
 - E.g., send, recv, allreduce, compute-for-a-period, wait, etc.

Firefly: Network traffic

- Purpose: Create network traffic, based on application communication patterns, at large scale
 - Enables testing the impact of network topologies and technologies on application communication at very large scale
- Scales to 1 million nodes
- Supports multiple “cores” per Node
 - Interaction between cores limited to message passing
- Supports space sharing of the network
 - Multiple “apps” running simultaneously

Firefly: Simulating large networks

- A network node consists of
 - Driver (the “application”)
 - NIC
 - Router
- Nodes are connected together via the routers to form the network
 - Fat tree, torus, etc.
- Firefly is the interface between the driver and the router
 - Message passing library → Firefly Hades
 - NIC → Firefly NIC



Scheduler

- Models HPC system-wide job scheduling
- Three components
 - **Sched:** schedules and allocates resources for a stream of jobs
 - **Node:** runs scheduled jobs on their allocated resources
 - **FaultInjection:** injects failures onto the resources
- The scheduler is currently a stand-alone element library
 - The schedComponent and nodeComponent must be used together
 - The faultInjectionComponent is optional